**HOSPITAL MANAGER ENGINEERING METHOD SPECIFICATION**

1. **Problem:**

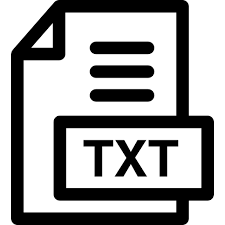
A medical institution has the need for an application used for the tracking of the patients in the facilities. For this purpose, it is required to have a database with the patients that has once been in the laboratory, whose information will then be then used to track them inside the facilities of the laboratory. Inside the laboratory there are several units, three will be assumed for this early stage of development, each one will have a separated waiting list which positions will be determined by the priority of each patient. The priority is to be determined by some attributes of the patient, those attributes are age and ailments. As this system will be connected to other systems in the future, it is necessary to simulate this by taking patients out of each queue every 2 minutes.

1. **Required knowledge:**
   1. **Data structures:** 
      1. **Hash table:** A hash table is a structure capable of storing information depending on a given “key” value, this value is then converted into a position on an array where the information is to be stored, which then can be retrieved with the same key, in average this operation takes θ(1) time.
      2. **Priority queue:** A queue that is reorganized depending on a certain attribute (priority) of its members. This kind of structure uses a heap to store and sort the data inside of it. A heap is a way of saving the elements in an array as if they were part of a graph in which on top there are the smallest elements, and on the bottom, there are the biggest ones.
      3. **Stack:** A stack is a structure conformed by a linked list that has one characteristic feature: the last one that enters is the first one that exits. This means that for each element inserted, it is only possible to access and delete the last element in the list.
      4. **Linked list:** A linked list is a data structure consisting of elements that point to each other rather than a holder pointing to all of them, each element must at least point to the element next to it.
   2. **Time taking in java:** There are multiple ways to measure time in java, these methods include:
      1. **Instant taking:** Different classes part of the JDK can take an instant at any moment, so, to know how much time has passed between two instants it possible to measure the difference between these two instants. One way of doing this is by using System.nanoTime(), this takes the number of nanoseconds that have passed after 00:00:00 January 1, 1970; as such, by having two instants, it is possible to get their difference to know how much time passed between two events.
      2. **Scheduled Executor Service (SES):** This is an interface that can schedule commands to run after a given delay or execute them periodically. This tool accepts relative delays and periods, but absolute times or dates. Knowing this, the “every 2 minutes pass the patient” requirement can be easily implemented since we can execute the “pass patient” method every 2 minutes with this interface.
2. **Possible solutions per requirement:**
   1. **R1:** 
      1. **Database solutions:**
         1. To save the patients locally, it is possible to save them in a .txt file that will then be read by the program every time it boots.
         2. To save the patients locally, it is possible to save them in a .json file that will be read every time the program boots.
      2. **In memory save solutions:** 
         1. To save the loaded patients, an option is to have an ArrayList with all the patients, as it will not be limited by a fixed size, it can have as many patients are necessary, each patient would count with an internal id corresponding to their position in the ArrayList.
         2. To save the loaded patients, an option is to have a binary search tree, as this would allow swift search for the patients that are expected to go more to the lab.
         3. To save the loaded patients, an option is to have a hash table with closed addressing that holds in each of its positions a linked list, as this would allow for fast insertion and extraction of data, which would not require the addition of an internal id for each patient, as their own id would serve as the key, but also have endless positions because of the linked list.
   2. **R2:**
      1. **Patient registration solutions:**
         1. Of a patient, the same information is always asked, therefore, there are not really any significant variations in the process of registration.
      2. **Search patient solutions:**

This solution depends on which registering patient solution we will choose.

* + - 1. To search for a patient, in all cases we must input their ID so the program will automatically return the patient object depending on which data structure we will use.
  1. **R3:** 
     1. **Priority differing queues solutions:**
        1. An ArrayList with each of the patients, where position zero will be considered the first in queue, and another ArrayList that holds the patients that have a priority other than zero, this is to be applied per unit of the lab.
        2. A queue with the patients that have no priority, and another that holds the patients with priority, this is to be applied per unit of the lab.
        3. A queue with the patients that have no priority, and a priority queue that holds priority patients depending on their priority, this is to be applied per unit of the lab.
        4. A single priority queue that places patients depending on their base priority, and that adds priority to each patient in queue after a patient has been called, this is to be applied per unit of the lab.
  2. **R4:**
     1. **Priority discerning:** 
        1. The attributes used for discerning the priority of a patient in queue will always be the same unless option four is chosen for the queuing system, where time in queue becomes the new attribute.
  3. **R5:**
     1. **Undo changes solutions:**
        1. An ArrayList containing the possible actualizable modules. When a change is made on one or more of the previous defined modules, a copy of the module(s) before they were changed will be stored in this ArrayList. A global variable will define the position of the ArrayList to recover in case of using the undo function.
        2. A Stack containing the possible actualizable modules. When a change is made on one or more of the previous defined modules, a copy of the module(s) before they were changed will be stored in this stack. In case of using the undo function, the stack will return the last change made.
  4. **R6:**
     1. The implementation of requirement six will only vary in design which it is not being disclosed in this document, as all that is pertinent to it are the implementations of the logic behind the program, not how it presents itself.
  5. **R7:**
     1. **Time management:**
        1. To take patients out of queue in simulation of a working laboratory, a time instant could me taken the moment a patient is in the first position of the queue, and another time instant could be taken the moment the user selects an option, where by this difference it could be discerned whether or not the patient has been in front for a given amount of time, then if the amount of time that passed is greater or equal to 2 minutes, then the patient is taken out of the queue, moreover, if the time that passed is enough for another patient to pass, then it will, this would happen until no more patients could have passed, then, for the next patient in queue, a new time instant is taken and in case the last patient’s time was greater than 2 minutes, but not enough for another patient to pass, then the time it went over is subtracted from the time instant taken.
        2. To take patients out of queue in simulation of a working laboratory we can use the scheduled executor service, and set a 2-minute period execution, which will call the pass patient method every time it reaches the 2 minutes.

1. **Idea discard:**
   1. **R1:**
      1. **Database solution:** 
         1. Neither idea is unviable enough to be discarded at a first glance.
      2. Texto

         Descripción generada automáticamente**R1.1:**
      3. **In memory save solution:**
         1. **Binary search tree:** In the situation proposed, a binary search tree is way too situational for it to be useful, as its main advantage, that it naturally organizes all data, is lost in the lack of necessity for organization found in the problem in question, and while it may be useful to know which patients come the most to the laboratory, it is not enough of an advantage to make this structure viable in the slightest.
      4. **R1.2:**
         1. Imagen que contiene Diagrama

            Descripción generada automáticamente**ArrayList design:**
         2. **Hash table design:**

Forma

Descripción generada automáticamente

* 1. **R2**
     1. **Patient registrations solutions:** 
        1. Neither of the ideas proposed is unviable enough to be discarded at a first glance.
     2. **Patient search solution:**
        1. There are not multiple options as the methos relies in its entirety on which data structure is chosen, as such, there is nothing to discard.
  2. **R3:**
     1. **Priority differing queue:**
        1. **ArrayList:** This solution is unviable as it is slow to add and extract patients from it, this means that the two main operations cannot be completed in a desirable time, making the solution undesirable. As for its only benefit, the fact that there is fast access to every member of the queue, it is worthless as there is never a need to access a random, known beforehand position of the queue, therefore, this solution is unviable.
        2. **Two regular queues:** While this solution does not have anything inherently wrong, it has one mayor flaw: it cannot discern based on priority levels. It is important to attend high risk patients as swiftly as possible, therefore, higher priority patients should be called first, even over lower priority patients, so, a merely arrival time-based queue is not good for this purpose, and because of that, it is discarded.
     2. **R3.1:**
        1. Imagen que contiene Diagrama

           Descripción generada automáticamenteImagen que contiene Texto

           Descripción generada automáticamenteTexto

           Descripción generada automáticamente con confianza media**Queue and priority queue design (this idea can be used for both queues):**
     3. **R3.2:**
        1. **Priority Queue design:**

Forma

Descripción generada automáticamente

* 1. **R4:**
     1. As there was only one viable solution found, there is nothing to discard here.
  2. **R5:**
     1. Both solutions are viable since they work in a remarkably similar way.
  3. **R6:**
     1. As this requirement only real variation is design related, it will not be addressed in this document.
  4. **R7:**
     1. Neither idea is unviable enough for there to be any early discard of either of them.

1. **Idea evaluation:**
   1. **R1:**
      1. **Database solution:** For a makeshift database, it was decided to evaluate two attributes: Ease of implementation (5 simplest, 1 hardest) and versatility (5 most versatile, 1 least versatile) The results from the evaluation are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ease of implementation | Versatility | Total |
| .txt | 4 | 4 | 8 |
| .json | 3 | 4 | 7 |

As shown above, both solutions are remarkably similar, but, as the .txt implementation is easier to implement, it is the chosen implementation.

* + 1. **In memory save solution:** For the data structure in which the database is to be loaded as the program is running, it was decided to evaluate three attributes: insertion speed (4 constant time, 3 logarithmical time, 2 lineal time, 1 polynomial time), extraction speed (4 constant time, 3 logarithmical time, 2 lineal time, 1 polynomial time) and ease of implementation (5 easiest, 1 hardest) The results are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Insertion speed | Extraction Speed | Ease of implementation | Total |
| ArrayList | 4 | 2 | 5 | 11 |
| Hash table | 4 | 4 | 4 | 12 |

As shown above, the main disadvantage of an ArrayList is the speed of the extraction of a particular value, this is because the user has no knowledge of the internal id of the ArrayList, and as such the whole list of patients must be checked to see if the id of the patient matches the id of the desired patient, meanwhile, as the key of the patient would be the id in the hash table, hashing would be the only thing necessary to find the patient, and therefore, in average, the fetching process of a patient would be performed in θ(1) time.

* 1. **R2:** 
     1. **Patient registration solution:** There are no alternative solutions, as such there is nothing to choose from.
     2. **Patient search solution:** The implementation of the search method relies on which data structure was chosen to hold the patients; therefore, it is of no interest in this discussion.
  2. **R3:**
     1. **Priority differing queues:** For this implementation, the attributes to be evaluated are Ease of implementation (5 easiest, 1 hardest), convenience for patients (5 most convenient, 1 least convenient), fairness of attention (5 fairest, 1 least fair) The results are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Ease of implementation | Convenience for the patients | Fairness of attention | Total |
| Queue/priority queue | 5 | 4 | 3 | 12 |
| Priority queue | 3 | 5 | 5 | 13 |

While harder to implement, the single priority queue is deemed to be better, this is because with a single, turn based priority queue it is ensured that patients will not be forever in queue as higher priority patients come into the queue, therefore, it ensures that while higher priority patients will be attended first, lower priority patients will be attended in a fair time.

* 1. **R4:**
     1. **Priority discerning:** There are no alternative solutions, as such there is nothing to choose from.
  2. **R5:**
     1. **Undo solution:** The attribute to be evaluated are insertion time (4 constant time, 3 logarithmical time, 2 lineal time, 1 polynomial time) and extraction time (4 constant time, 3 logarithmical time, 2 lineal time, 1 polynomial time) The results are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Insertion time | Extraction time | Total |
| ArrayList | 2 | 4 | 6 |
| Stack | 4 | 4 | 8 |

Constantly adding data into an ArrayList is slower compared to other addition operations, as the addition of data requires the creation of an array of length n+1 where every position except the position n contains the data from the previous array, therefore, there is the need to assign every value to the new array and that sets the speed of this operation at O(n). On the other hand, a stack’s addition operation is as simple as replacing the head with a new one, as such, it is fast, and while the stack has the disadvantage that there is only one accessible node in constant time, which is all that is needed for this operation, and hence this is the chosen solution.

* 1. **R6:**
     1. This is a design intended solution, so there is nothing to choose.
  2. **R7:**
     1. **Time management solution:** The attribute to be evaluated that was found relevant is Ease of implementation (5 easiest, 1 hardest)

The results are as follows:

|  |  |  |
| --- | --- | --- |
|  | Ease of implementation | Total |
| Nano Time | 4 | 4 |
| SES | 5 | 5 |

The first option considered was using the nano time solution, as it was thought to be the best option. Nano time can be used to compare if two minutes have passed and if it is true, it will pass the patient. But after research it was found that there is an interface that can be used to schedule and execute an action in each period. The implementation will be quite simple as we only must set the period between executions and the desired action to execute, in this case, the pass patient action.

**Bibliography**

Docs.oracle.com. 2022. *ScheduledExecutorService (Java Platform SE 7 )*. [online] Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html> [Accessed 16 October 2022].